Inheritance in the Presence of Asynchronous Method Calls

Einar Broch Johnsen and Olaf Owe
Department of Informatics, University of Oslo
PO Box 1080 Blindern, N-0316 Oslo, Norway
{einarj,olaf}@ifi.uio.no

Abstract— **This paper considers a formal object-oriented model for distributed computing. Object orientation appears as a leading framework for concurrent and distributed systems. However, the synchronization of the RPC communication model is unsatisfactory in many distributed systems. Asynchronous message passing gives better control and efficiency in this setting, but lacks the structure and discipline of traditional object-oriented methods. The integration of the message concept in the object-oriented paradigm is unsettled, especially with respect to inheritance and redefinition.**

We propose an approach combining asynchronous method calls and conditional processor release points, which reduces the cost of waiting for replies in the distributed environment while avoiding low-level synchronization constructs such as explicit signaling. Even the lack of replies to method calls in unstable environments need not lead to deadlock in the invoking objects. This property seems attractive in asynchronous, open, or unreliable environments. Furthermore, the approach allows active and passive behavior (client and server) to be combined in concurrent objects in a very natural way.

In this paper, we consider the integration of these constructs with a mechanism for multiple inheritance within a small object-oriented language. The language constructs are formally described by an operational semantics defined in rewriting logic.

I. Introduction

The importance of inter-process communication is rapidly increasing with the development of distributed computing, both over the Internet and over local networks. Object orientation appears as a leading framework for concurrent and distributed systems, and has been recommended by the RM-ODP [1], but object interaction by means of method calls is usually synchronous. The mechanism of remote procedure calls (RPC) [2] has been derived from the setting of sequential systems, and works well for tightly coupled systems. It is clearly less suitable in a distributed setting where the components are loosely coupled. Here synchronous communication gives rise to undesired and uncontrolled waiting, and possibly deadlock. Asynchronous message passing gives better control and efficiency, but does not provide the structure and discipline inherent in method declarations and calls.

Intuitive high-level programming constructs are needed to unite object orientation and distribution in a natural way. We do not believe that distribution should be transparent to the programmer as in the RPC model, rather communication in the distributed setting should be *explicitly asynchronous*. Separating execution threads from objects breaks the modularity and encapsulation of object orientation, leading to a very low-level style of programming. Models of distributed systems based on asynchronously communicating concurrent objects seem much more natural. This paper considers programming constructs for concurrent objects, based on communication by *asynchronous method calls* and a notion of *processor release points*. Processor release points are used to influence the implicit internal control flow in concurrent objects. Objects have an associated processor and a mechanism for scheduling of pending processes. This reduces time spent waiting for replies to method calls in a distributed environment and allows objects to dynamically change between active and reactive behavior (client and server). These notions were formalized with the operational semantics of the Creol language [3].

This paper considers how these notions may be integrated with the structuring mechanism provided by inheritance, addressing high-level program constructs relevant to the integration of object orientation and distribution. In contrast to related work, we propose an integration of asynchronous communication and inheritance which allows method overloading and redefinition. To illustrate the generality of the approach, multiple inheritance is considered. To explain the approach and motivate its suitability, the proposed language constructs for distributed object systems are integrated in the object-oriented Creol language with a simple operational semantics, while maintaining the efficiency control of asynchronous message passing. The operational semantics of the language extension is defined in rewriting logic [4], extending Creol's operational semantics which is executable as a language interpreter in the tool Maude [5]. Our experiments suggest that rewriting logic and Maude provide a well-suited platform for experimentation with language constructs and concurrent environments.

Paper overview: Sect. II outlines the overall setting of the approach. Sect. III extends the Creol language with inheritance. Sect. IV gives some examples. Sect. V gives a formal, operational semantics for the language. Sect. VI considers related work and Sect. VII concludes the paper.

II. An Approach to Object-Oriented Distributed Systems

A. Asynchronous method calls

According to the RM-ODP, distributed components may be seen as (collections of) objects that run in parallel and communicate by means of remote method calls. However, existing interaction models do not combine the method concept with distributed concurrent objects in a satisfactory manner. The three basic interaction models for concurrent processes [2] are shared variables, RPC, and message passing. As shared memory models do not generalize well to distributed environments, shared variables are discarded. With the RPC model, an object is activated by a method call. Control is transferred with the call so there is a master-slave relationship between the caller and the callee. A similar approach is taken with the execution threads of e.g. Hybrid [6] and Java [7], and concurrency is achieved through multithreading. The interference problem for shared variables reemerges when threads operate concurrently in the same object, which happens with non-serialized methods in Java. Reasoning about programs in this setting is a highly complex matter [8], [9]: Safety is by convention rather than by language design [10]. Verification considerations therefore suggest that all methods should be serialized as in e.g. Hybrid. Restricting to serialized methods, the invoking process must *wait* for the return of a call, blocking for any other activity in the object. In a distributed setting this limitation is severe; delays and instability may cause much unnecessary waiting. A nonterminating method will also block evaluation of other method calls, which makes it difficult to combine active and passive behavior in the same object.

In contrast, message passing does not transfer control. For synchronous message passing, as in Ada's Rendezvous mechanism, both sender and receiver must be ready before communication can occur. Method calls may be modeled by pairs of messages, on which the two objects must synchronize [2]. For distributed systems, this synchronization still results in much waiting. In the asynchronous setting, messages may be emitted even when the receiver is not ready. Communication by asynchronous message passing is well-known from e.g. the Actor model [11], [12]. Generative communication in Linda [13] is an approach between shared variables and asynchronous message passing, where messages without an explicit destination address are shared on a possibly distributed blackboard. However, method calls imply an ordering on communication not easily captured in the Actor model and Linda. We believe that a satisfactory notion of method calls for the distributed setting should be asynchronous, combining the advantages of asynchronous message passing with the structuring mechanism provided by the method concept.

B. Inheritance and structuring mechanisms

Inheritance in object-oriented languages basically serves two purposes. First, class inheritance is a powerful structuring mechanism for code reuse. Class extension and method redefinition are convenient both for development and understanding of code. Calling superclass methods in a subclass method enables reuse in redefined methods, making the relationship between the method versions explicit. Thus, this facility is clearly superior to cut-and-paste programming with regard to the ease with which existing code may be inspected and understood. Second, inheritance can be understood in terms of reasoning reuse, obeying the *substitutability* principle: As a subclass is a specialization of a superclass, an object of the subclass may replace an object of the superclass. This has led to an active field of research on behavioral subtyping [14], [15], which aims at identifying conditions for safe substitutability. Although many languages identify the subclass and subtype relations, in particular with regard to parameter passing, several authors argue that inheritance relations for code and for behavior should be kept distinct. Identifying the two relations leads to severe restrictions on code reuse which may seem unattractive to programmers [16].

In order to solve the conflict between unrestricted code reuse in subclasses, and behavioral subtyping and incremental reasoning control [15], [16], we use behavioral interfaces [17], [18] to type object variables and remote calls, and allow multiple inheritance at both the interface and class level. Interface inheritance is restricted to a form of behavioral subtyping [15], whereas class inheritance may be used freely. In this paper, interfaces are given a purely syntactic presentation.

Inherited class (re)declarations are resolved by disjoint union combined with an ordering of the super classes. A class may implement several interfaces, provided that it satisfies the syntactic and semantic requirements stated in the interfaces. An object of class C supports an interface I if the class C implements I. Reasoning control is ensured by substitutability at the level of interfaces: *an object supporting an interface I may be replaced by another object supporting I or a subinterface of* I in a context depending on I, although the latter object may be of another class. Subclassing is unrestricted in the sense that implementation claims (as well as class invariants) are not in general inherited at the class level.

With distinct inheritance and subtyping hierarchies, it is possible to inherit only a subset of the attributes and methods of a superclass. However, this would require considerable work establishing invariants for parts of the superclass that appear desirable for inheritance, either anticipating future needs or while designing subclasses. The *encapsulation principle* for class inheritance states that it should suffice to work at the subclass level to ensure that the subclass is well-behaved when inheriting from a superclass: Code design as well as new proof obligations should occur in the subclass only. Situations that break the encapsulation principle have been labeled inheritance anomalies [19], [20], in which reuse requires redefinition. Reasoning considerations therefore suggest that all attributes and methods of a superclass are inherited, but method redefinition may violate the semantic requirements of an interface.

C. Asynchronous method calls and inheritance

Distributed communication based on asynchronous message passing does not offer the structuring mechanisms provided by method definitions. Notions of asynchronous methods may be build on top of asynchronous communication paradigms such as Actors and Linda, fixing a method as either synchronous or asynchronous. However, formalisms taking this approach have traditionally either not supported inheritance [21], [22], imposed redefinition of asynchronous methods [23], or used inheritance as a means to introduce nondeterminism in the language [5], [24]. Inheritance in the object-oriented sense has not been supported by these formalisms. In particular, traditional method redefinition and overriding have not been available. In this paper, we propose an approach which allows methods to be invoked in either a synchronous or an asynchronous manner and which combines (asynchronous) methods calls with inheritance, allowing redefinition as well as overriding.

III. An Overview of Creol

This section proposes programming constructs for distributed concurrent objects, based on asynchronous method calls, processor release points, and multiple inheritance. Concurrent objects are potentially active, encapsulating execution threads; consequently, elements of basic data types are not considered objects. In this sense, our objects resemble top-level objects in e.g. Hybrid. Objects have identity: communication takes place between named objects and object identifiers may be exchanged. As motivated above, Creol objects are typed by interfaces, resembling CORBA's IDL, but extended with semantic requirements and mechanisms for type control in dynamically reconfigurable systems. Strong typing implies that invoked methods are supported by the called object (when not null), and formal and actual parameters match.

A. Interfaces and strong typing

Two kinds of variables are declared; an object variable typed by an interface and an ordinary variable typed by a data type. We assume a common type Data of basic data values, such as the natural numbers Nat, strings Str, and object identifiers Obj, including *this*, which may be passed as arguments to methods. Expressions Expr evaluate to Data. Denote by Var the set of program variables, by Mtd the set of method names, and by Label the set of method call identifiers. Object variables are declared with Expr values, which evaluate to data in the context of the actual class parameters. In order to focus the discussion on asynchronous method calls, processor release points, and inheritance in this setting, standard typing issues will not be discussed in further detail in this paper.

Strong typing ensures that for each method invocation $o.m(\mathit{In};\ \mathit{Out})$, where I is the declared interface of o, the actual object o (if not null) will support I and the method m will be understood. As object variables are typed by interfaces, only the methods mentioned in the interface (or its super-interfaces) are visible. Interfaces do not specify instance variables, so these cannot be directly referenced. Explicit hiding of class attributes and methods is not needed. Interfaces describe viewpoints to objects and have the following general form:

```
interface $F$ ($\langle$parameters$\rangle$) inherits $F_1, F_2, \ldots, F_m$
begin with $G$
    op $m_1(\ldots)$
      $\ldots$
    op $m_n(\ldots)$
end
```

where $F, F_1, \ldots, F_m$, and G are interfaces. Interfaces may have both value and object parameters, typed respectively by data types and interfaces. Interface parameters describe the minimal environment that any object offering the interface needs at the point of creation.

For active objects we may want to restrict invocation access to objects of a particular interface. This way, the active object can invoke methods of the caller and not only passively complete invocations of its own methods. Use of the **with** clause restricts the communication environment of an object, as considered through the interface, to external objects offering a given *cointerface* [17], [18]. For some objects no such knowledge is required, which is captured by the keyword *Any* in the **with** clause. Mutual dependency is specified if two interfaces have each other as cointerface.

Example. We consider the interfaces of a node in a peer-to-peer file sharing network. A *Client* interface captures the client end of the node, available to any user of the system. It offers methods to list all files available in the network, and to request the download of a given file from a given server. A *Server* interface offers a method for obtaining a list of files available from the node, and a mechanism for downloading packs, i.e. parts of a target file. The Server interface is available to other servers in the network. A *Client2* interface is only available to Servers with a method to fetch a list of trusted servers from the client.

```
interface Client     interface Server     interface Client2
begin with Any       begin with Server    begin with Server
  op availFiles        op listFiles         op getServers
  op reqFile           op getLength       end
end                    op getPack
                     end
```

The **with**-construct allows the typing mechanism to deduce that any caller of a server request will understand the *listFiles* and *getPack* methods. To save space, discussion of method parameters is postponed to Sect. IV. The two interfaces may be inherited by a third interface *Peer*, describing nodes able to act according to both the client role and the server role:

```
interface Peer inherits Client, Client2, Server
begin end
```

B. Class Declarations with Multiple Inheritance

At the imperative level, attributes (class variables) and method declarations are organized in classes, which may have value and object parameters similar to interface parameters. We consider multiple inheritance where all attributes and methods of a superclass are inherited by the subclass, and where superclass methods may be redefined. Class inheritance is declared in Creol by a keyword **inherits** which takes as its argument an *inheritance list*, i.e. a list of class names $C(\mathrm{E})$ where E provides actual class parameters. Say that a method is defined *above* a class C if it is declared in C or in at least one of the classes inherited by C. When a method is invoked in an object o of class C, a method body is identified in the inheritance graph and bound to the call. In order to keep the exposition simple, the method call will be bound to the first possible method definition above C in the inheritance graph, in a left-first depth-first order, and we will here ignore the types of the method parameters in the binding strategy. (In Creol typing considerations are made to ensure strong typing.)

The encapsulation provided by interfaces suggests that external calls to an object of class C are virtually bound to the closest method definition above C. However, the object may internally invoke methods of its superclasses. In the setting of multiple inheritance and overloading, methods defined in a superclass may be accessed in the subclass by qualified references. We let attributes of the superclass be accessed in the same way. Consequently, identically named attributes which are inherited from several superclasses are only identified if they come from a common ancestor class.

Objects are dynamically created instances of classes. Object attributes are encapsulated and can only be accessed via the object's methods. Among the declared methods, we distinguish the *run* method, which is given special treatment operationally. After initialization, the *run* method, if provided, is started. Methods may be invoked internally and by other objects of appropriate interfaces. When called from other objects, methods reflect passive or reactive behavior in the object, whereas *run* reflects active behavior. Methods need not terminate and all method instances may be temporarily *suspended*.

C. Methods Declarations

1) Asynchronous Methods: An object offers methods to its environment, specified through a number of interfaces and cointerfaces. All interaction with an object happens through method calls. In the asynchronous setting method calls can always be emitted, because the receiving object cannot block communication. *Method overtaking* is allowed: if methods offered by an object are invoked in one order, the object may start the method instances in another order. A method instance is, roughly speaking, program code with inner processor release points, evaluated in the context of local variables.

Different method executions may be interleaved, so the values of an object's program variables are not entirely controlled by a method instance with a release point. Therefore, a method may have local variables supplementing the object variables. In particular, the values of formal parameters are stored locally, but other local variables may also be created. Semantically, an instantiated method is represented by a *process* $\langle \mathrm{S}, \mathrm{L} \rangle$ where S is a sequence of commands and $\mathrm{L} : \mathsf{Var} \to \mathsf{Data}$ the local state. Consider an object o which offers the method

op $m($**in** $n : \mathsf{Nat}$ **out** $d : \mathsf{Data}) ==$ **var** $i : \mathsf{Nat}{=}\ 0;\ \mathrm{S}$.

to the environment. Syntactically, method declarations end with a period. Accepting a call to m with argument 2 from another object o' creates a process $\langle \mathrm{S}, \{\mathit{label} \mapsto t, \mathit{caller} \mapsto o', n \mapsto 2, d \mapsto \mathit{nil}, i \mapsto 0\}\rangle$ in the object o. An object may have several (suspended) instances of the same method, possibly with different values for local variables. The local variables *label* and *caller* are reserved to identify the call and the caller for the reply, which is automatically emitted at method termination, i.e. when computation of S is completed.

An asynchronous method call is made with the command $t!x.m(\mathrm{E})$, where $t \in \mathsf{Label}$ provides a locally unique reference to the call, x is an object expression, m a method name, and E an expression list with the actual parameters supplied to the method. Labels identify replies, and may be omitted if a reply is not explicitly requested. As no synchronization is involved, process execution can proceed after calling an external method until the return value is actually needed by the process. Return values from the call are explicitly fetched, say in a variable list V, by the command $t?(\mathrm{V})$. This command treats V as a future variable [21]: If a reply has arrived, return values are assigned to V and execution continues without delay. Otherwise, process execution is blocked. In order to avoid blocking in the asynchronous case, processor release points are introduced for reply requests (Sect. III-C.2): If no reply has arrived, execution is *suspended*.

The syntax $x.m(\mathrm{E}; \mathrm{V})$, where the semicolon separates input expressions from output variables, is adopted for synchronous (RPC) method calls, immediately blocking the processor while waiting for the reply. The language does not support monitor reentrance, mutual synchronous calls may therefore lead to deadlock. In order to execute local calls, the invoking process must eventually suspend its own execution. In particular, execution of synchronous local calls will precede the active code. Local calls need not be prefixed by an object identifier, in which case they may be identified syntactically, otherwise equality between caller and callee is determined at runtime.

2) Inner Processor Release Points: Guarded commands g are used to explicitly declare potential processor release points **await** g. Guarded commands can be nested within the same local variable scope, corresponding to a series of processor release points. When an inner guard which evaluates to false is encountered during process execution, the process is suspended and the processor released. After processor release, any suspended process may be selected for execution.

The type Guard is constructed inductively:

- $\mathit{wait} \in \mathsf{Guard}$ (explicit release)
- $t? \in \mathsf{Guard}$, where $t \in \mathsf{Label}$
- $b \in \mathsf{Guard}$, where b is a boolean expression over local and object state
- $g_1 \wedge g_2$ and $g_1 \vee g_2$, where $g_1, g_2 \in \mathsf{Guard}$.

Use of *wait* will explicitly release the processor. The reply guard $t?$ succeeds if the reply to the method invocation with label t has arrived. Evaluation of guards is done atomically. We let **await** $g \wedge t?(\mathrm{V})$ abbreviate **await** $g \wedge t?; t?(\mathrm{V})$ and **await** $p(\mathrm{E}; \mathrm{V})$ abbreviate $t!p(\mathrm{E})$; **await** $t?(\mathrm{V})$ for some fresh label t.

Internal control flow in objects is expressed by composing guarded commands. Let GS_1 and GS_2 be guarded commands **await** $g_1; \mathrm{S}_1$ and **await** $g_2; \mathrm{S}_2$. Inner guards are obtained by sequential composition; in the statement $GS_1; GS_2$, the guard g_2 is a potential release point. Non-deterministic choice is expressed by $GS_1 \Box GS_2$, which may compute S_1 if g_1 evaluates to true or S_2 if g_2 evaluates to true. Non-deterministic merge is expressed by $GS_1 \interleave GS_2$, defined as $(GS_1; GS_2) \Box (GS_2; GS_1)$. *Synchronized merge*, $GS_1 \,\&\, GS_2$, is defined as **await** $g_1 \wedge g_2; \mathrm{S}_1; \mathrm{S}_2$, treating non-guarded arguments as guarded by true and expanding synchronized method calls (see Sect. IV-B). Control flow without potential processor release uses **if** and

Syntactic categories.	*Definitions.*
g in Guard	$g ::= \textit{wait} \mid b \mid t? \mid g_1 \wedge g_2 \mid g_1 \vee g_2$
p in MtdCall	$p ::= x.m \mid m@\textit{classname} \mid m$
S in StmList	$\mathrm{S} ::= s \mid s; \mathrm{S}$
s in Stm	$s ::= \textbf{skip} \mid (\mathrm{S})$
t in Label	$\mid \mathrm{S}_1 \Box \mathrm{S}_2 \mid \mathrm{S}_1 \Vert \mathrm{S}_2 \mid \mathrm{S}_1 \,\&\, \mathrm{S}_2$
v in Var	$\mid \mathrm{V} := \mathrm{E} \mid v := \textbf{new}\ \textit{classname}(\mathrm{E})$
e in Expr	$\mid \textbf{if}\ b\ \textbf{then}\ \mathrm{S}_1\ \textbf{else}\ \mathrm{S}_2\ \textbf{fi}$
x in ObjExpr	$\mid \textbf{while}\ b\ \textbf{do}\ \mathrm{S}\ \textbf{od}$
b in BoolExpr	$\mid t!p(\mathrm{E}) \mid !p(\mathrm{E}) \mid p(\mathrm{E};\mathrm{V}) \mid t?(\mathrm{V})$
m in Mtd	$\mid \textbf{await}\ g \mid \textbf{await}\ g \wedge t?(\mathrm{V}) \mid \textbf{await}\ p(\mathrm{E};\mathrm{V})$

Fig. 1. An outline of the language syntax for method definitions, with typical terms for each category. Capitalized terms such as E denote lists, sets, or multisets of the given syntactic categories, depending on the context.

while constructs, and assignment to local and object variables is expressed as $\mathrm{V} := \mathrm{E}$ for a disjoint list of program variables V and an expression list E, of matching types. While expressions are without side effects, **new** creates a new object in the environment and returns its object identifier. In-parameters as well as *this*, *label*, and *caller* are read-only variables.

With inner release points, the object need not block while waiting for replies. This approach is more flexible than future variables: suspended processes or new method calls may be evaluated while waiting. If the called object never replies, deadlock is avoided as other activity in the object is possible. However, when the reply arrives, the *continuation* of the process must compete with other enabled suspended processes.

For code inside a subclass of C,we introduce the syntax $t!m@C(\mathit{In})$ for asynchronous and $m@C(\mathit{In};\mathit{Out})$ for synchronous *local* invocation of a method above C in the inheritance graph. As the binding of such calls may be done without knowing the class of *this* object, they are called *static*, in contrast to calls without @, called *virtual*. As objects are typed by interfaces, external calls are always virtual. Fig. 1 summarizes the language syntax.

IV. Examples

A. Peer-to-peer Networking

In a distributed peer-to-peer file sharing system, servers may arrive and disappear dynamically. A client requests a file from a server in the distributed network, and downloads it as a series of packet downloads until the file download is complete. The connection to the server may be blocked, in which case the download will automatically resume if the connection is reestablished. A client may run several downloads concurrently, at different speeds. We assume that every node in the network has an associated database with shared files. Downloaded files are stored in this database, which is not modeled here but implements an interface *DB*:

```
interface DB
begin with Server
  op getFile(in fId:Str out file: List[List[Data]])
  op getLength(in fId:Str out length:Nat)
  op storeFile(in fId:Str, file:List[Data])
  op listFiles(out fList:List[Str])
end
```

The method *getFile* returns a list of packets, i.e. a sequence of sequences of data, for network transmission, *getLength* returns the number of such packets for a given file name, *listFiles* returns the list of available files, and *storeFile* adds a file to the database, possibly overwriting an existing file.

The class *ServerCl* takes object parameters of interfaces *DB* and *Client*, and implements the *Server* interface. The parameters provide static links to the local database and client. The latter decides which remote servers may be trusted for downloading files.

```
class ServerCl (myClient: Client2, myDB:DB) implements Server
begin with Server
  op getLength(in fId:Str out lth:Nat) ==
    await myDB.getLength(fId;lth) .
  op getPack(in fId:Str, pNbr:Nat out pack:List[Data]) ==
    var f:List[Data]; await myDB.getFile(fId;f); pack:=f[pNbr] .
  op listFiles(out servers: List[Str], files: List[Str]) ==
    await myClient.getServers( ; servers);
      await myDB.listFiles( ; files) .
end
```

The method *getLength* returns the number of packs for a given file, *getPack* a particular pack in the transmission of a file, and *listFiles* the lists of known servers and available files, We let $s[i]$ be the i'th element of list s (for $0 \leq i \leq \mathit{length}(s)$). Note that *ServerCl* objects can have *several interleaved activities*: several downloads may be processed simultaneously as well as uploads to other servers, etc. All method calls are asynchronous: If a server temporarily becomes unavailable, the transaction is suspended and may resume at any time after the server becomes available again. Processor release points ensure that the processor will not be blocked in this case and transactions with other servers are not affected.

The class *ClientCl* takes an object parameter of interface *Server* and implements the *Client* interface:

```
class ClientCl (myServer:Server) implements Client, Client2
begin var trusted: List[Str] := myServer
with Server
  op getServers(out sList: List[Str]) == sList := trusted .
with Any
  op availFiles (out files:List[Str×Str]) == await aux(0; files) .
  op aux (in i:Nat  out files:List[Str×Str]) ==
    var t1, t2:Label, fList1: List[Str]; fList2: List[Str×Str];
     files := ε; if (i = length(trusted)) then skip else
       t1 ! trusted[i] . listFiles(); t2 ! this . aux(i+1);
      (await t1?(sList, fList1); trusted := trusted (sList\trusted);
        files := files; pair(trusted[i],fList1))
      ||| (await t2?(fList2); files := files fList2 ) fi .
  op reqFile(in sId:Str, fId:Str) ==
    var file, pack: List[Data], lth: Nat ;
      await sId.getLength(fId; lth);
    while (lth > 0) do await sId.getPack(fId, lth; pack);
        file:=(file; pack);lth:=lth - 1 od; !myDB.storeFile(fId,file) .
end
```

We denote by ε the empty list and by ';' list concatenation. For $t{:}T$ and $s, s'{:}\,\mathsf{List}[T]$, let $s \backslash s'$ be the list of elements in list s which do not occur in list s' and *pair*(t, s) the list of pairs obtained by a pairwise mapping of t onto s.
The method *availFiles* uses an auxiliary method *aux* returns a list of pairs where each pair contains a file identifier *fId* and the server identifier *sId* where *fId* may be found, and *reqFile* returns the file associated with *fId*. Note that the auxiliary function is *private* as it is not declared in the *Client* interface.

Nodes in the peer-to-peer network which implement the *Peer* interface can be modeled by the class *Node* below.

```
class Node (db:DB) inherits ServerCl(this, db), ClientCl(this)
   implements Peer
begin end
```

Due to the instantiation of the superclass parameters with *this*, several of the asynchronous calls considered above have now become local calls to the objects itself. Using inner release points, this does not cause any difficulties; asynchronous calls may be evaluated whenever the object is idle.

B. Inheriting synchronization constraints

We now demonstrate the use of Creol on examples from the literature on the inheritance anomaly [19], in particular anomalies related to the use of guards. Note that inheritance anomalies also occur in languages with single inheritance. Interfaces are omitted here, as they are not central to the discussion. Let *Buf* be a class with parameter *length*:Nat, unguarded operations *put*$(x{:}\,\mathsf{Data})$ and *get*$(\mathbf{out}\ x{:}\,\mathsf{Data})$, and an internal attribute *size* recording the current number of elements in the buffer. By means of the @ construct, we may easily add guards to make users wait when the operations cannot be performed properly:

```
class Buf1(length: Nat) inherits Buf(length)
begin with Any
  op put (in x: Data) == await size < length; put@Buf(x) .
  op get (out x: Data) == await size > 0; get@Buf(;x) .
  op get2 (out x1, x2: Data) == await size > 1;
        get@Buf(;x1); get@Buf(;x2) .
end
```

Here, we have added a *get2* operation where the guard ensures that two synchronous *get* calls can be performed properly.

We then consider the problems of *history sensitive behavior*, adding an operation *gget* that should behave like *get* expect that it must wait after a normal *get*. We first define a mix-in class *Lock*, with general synchronization operations:

```
class Lock
begin var locked: Bool=false .
with Any
  op unlock == locked := false .
  op lock == locked := true .
  op sync == await (¬ locked) .
end
```

We may now use multiple inheritance to add a lock to the buffer class, and redefine the buffer operations, adding synchronization by means of *synchronous merge*:

```
class Buf2(lth: Nat) inherits Buf1(lth), Lock
begin with Any
  op put (in x: Data) == unlock@Lock & put@Buf1(x) .
  op get (out x: Data) == lock@Lock & get@Buf1(;x) .
  op gget(out x: Data) == sync@Lock & get@Buf1(;x) .
end
```

We have obtained a history sensitive version of the buffer class by combining the two superclasses in a clean manner. The resulting *gget* is guarded by $(\neg\,\text{locked} \wedge \text{size} > 0)$, ensuring that both guards are satisfied before the operation may start. This is in general crucial to avoid deadlock, for instance if the *sync* operation grabs the lock (a *gget* would then block a succeeding *gget*):

```
op sync == await not locked; locked := true.
```

This reuse of inherited operations by synchronous merge and synchronous calls to superclass methods is semantically clean, e.g. partial correctness reasoning about s_1 and s_2 carries over to $s_1 \,\&\, s_2$ when any common program variables are not changed in neither statement. As Creol gives read only access to in-parameters and *this*, the requirement is guaranteed for synchronized merge of super operations from disjoint superclasses, $m_1@C_1(\ldots) \,\&\, m_2@C_2(\ldots)$, as in the above example. Consequently, synchronized merge guarantees maintenance of superclass invariants when used in this way [25].

This example shows how business code and synchronization code can be developed independently in Creol, and the two kinds of code can be combined effectively and cleanly. In contrast to recent aspect oriented approaches [20], including synchronization patterns and composition filters, we use the same basic language to express both kinds of code.

V. An Operational Semantics for Creol

The operational semantics of Creol is defined using rewriting logic [4]. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Each rewrite rule describes how a part of a configuration can evolve in one transition step. If rewrite rules may be applied to non-overlapping subconfigurations, the transitions may be performed in parallel. Consequently, concurrency is implicit in rewriting logic (RL). A number of concurrency models have been successfully represented in RL [4], [5], including Petri nets, CCS, Actors, and Unity, as well as the ODP computational model [26]. RL also offers its own model of object orientation [5].

Informally, a state configuration is a multiset of terms of given types. Types are specified in (membership) equational logic (Σ, E), the functional sublanguage of RL which supports algebraic specification in the OBJ [27] style. When modeling computational systems, configurations may include the local system states. Different parts of the system are modeled by terms of the different types defined in the equational logic.

RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules, supplementing the equations which define the term language. Assuming that all terms can be reduced to normal form, rewrite rules transform terms modulo the defining equations of E. Conditional rewrite rules are allowed, where the condition is formulated as a conjunction of rewrites and equations which must hold for the main rule to apply:

$$\textit{subconfiguration} \longrightarrow \textit{subconfiguration}\ \textbf{\textit{if}}\ \textit{condition}.$$

Rules in RL may be formulated at a high level of abstraction, closely resembling a compositional operational semantics. In fact, structural operational semantics can be uniformly mapped into RL specifications [28].

1) System Configurations: An asynchronous method call will be reflected by a pair of messages, and object activity will be organized around a *message queue* which contains incoming messages and a *process queue* which contains suspended processes, i.e. remaining parts of method instances. In order to increase parallelism in the model, message queues will be external to object bodies. A state configuration is a multiset combining Creol objects, classes, messages, and queues. As usual in RL, the associative constructor for lists, as well as the associative and commutative constructor for multisets, are represented by whitespace.

In RL, objects are commonly represented by terms of the type $\langle O : C \,| a_1 : v_1, \ldots, a_n : v_n\rangle$ where O is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's are the corresponding values [5]. We adopt this form of presentation and define Creol objects, classes, and external message queues as RL objects. Omitting RL types, a Creol object is represented by an RL object $\langle \mathit{Ob} \mid \mathit{Cl}, \mathit{Pr}, \mathit{PrQ}, \mathit{Lvar}, \mathit{Att}, \mathit{Lab}\rangle$, where *Ob* is the object identifier, *Cl* the class name, *Pr* the active process code, *PrQ* a multiset of suspended processes with unspecified queue ordering, and *Lvar* and *Att* the local and object state, respectively. Let τ be a type partially ordered by $<$, with least element 1, and let $\mathit{Next} : \tau \rightarrow \tau$ be such that $\forall x \,.\, x < \mathit{Next}(x)$. *Lab* is used to generate method call identifiers and values of type τ. Thus, the object identifier *Ob* and the generated local label value provide a globally unique identifier for each method call. Message queues are RL objects $\langle \mathit{Qu} \,|\, \mathit{Ev}\rangle$, where *Qu* is the queue identifier and *Ev* a multiset of unprocessed messages. Each message queue is a distinct term in the state configuration, associated with one specific Creol object.

The classes of Creol are represented by RL objects $\langle \mathit{Cl} \,|\, \mathit{Inh}, \mathit{Att}, \mathit{Mtds}, \mathit{Tok}\rangle$, where *Cl* is the class name, *Inh* is the inheritance list, *Att* a list of attributes, *Mtds* a multiset of methods, and *Tok* is an arbitrary term of sort τ. When an object needs a method, it is bound to a definition in the *Mtds* multiset of its class or of a superclass.

To pave the way for dynamic reconfiguration mechanisms, such as a dynamic class construct [29], the inheritance graph will not be statically given. We then need a binding mechanism which dynamically inspects the current class hierarchy as present in the configuration. As rewriting logic targets local change, there is no way to access all classes in a configuration in a single equation or rule. A natural solution is to use a *bind* message to be sent from a class to its superclasses, resulting in a *bound* message sent back to the object generating the *bind* message. To simplify the presentation we do not discuss the influence of parameter types on the binding mechanism.

In RL's object model [5], classes are not represented explicitly in the system configuration. This leads to ad hoc mechanisms to handle object creation, which we avoid by explicit class representation. The command *new* $C(\mathit{args})$ creates a new object with a unique object identifier, object variables as listed in the class parameter list and in *Att*, and places the code from the *run* method in *Pr*.

2) Concurrent Transitions: Concurrent change is achieved in the operational semantics by applying concurrent rewrite steps to state configurations. There are four different kinds of rewrite rules:

- *Rules that execute code from the active process:* For every program statement there is at least one rule. For example, the assignment rule for the program `V := E` binds the values of the expression list `E` to the list `V` of local and object variables.
- *Rules for suspension of the active process:* When an active process guard evaluates to false, the process and its local variables are suspended, leaving *Pr* empty.
- *Rules that activate suspended processes:* When *Pr* is empty, suspended processes may be activated. When this happens, the local state is replaced.
- *Transport rules:* These rules move messages into and out of the external message queue. Because the external message queue is represented as a separate RL object, it can belong to another subconfiguration than the object itself and it can therefore receive messages in parallel with other activity in the object.

When auxiliary functions are needed in the semantics, these are defined in equational logic, and are evaluated in between the state transitions [4]. The rules for the basic constructs concerning method calls, replies, guarded commands, local synchronous method calls, and creation of new objects, are now considered in more detail.

3) Method Calls: In the operational semantics, objects communicate by sending messages. Two messages are used to encode a method call. If an object o_1 calls a method m of an object o_2, with arguments *In*, and the execution of $m(\mathit{In})$ results in the return values *Out*, the call is reflected by two messages $\mathit{invoc}(o_2, m, (n\ o_1\ \mathit{In}))$ and $\mathit{comp}(n, o_1, \mathit{Out})$, which represent the invocation and completion of the call, respectively. In the asynchronous setting, invocation messages must include the caller's identity, so completions can be transmitted to the correct destination. Objects may have several pending calls to another object, so the completion message includes a locally unique label value n, generated by the caller.

When an object calls an external method, a message is placed in the configuration. The rewrite rule for this transition can be expressed as follows, ignoring irrelevant attributes in the style of Full Maude [5]:

$\langle O : Ob \mid Pr : (t!x.m(In); \textsc{s}), Lvar : \textsc{l}, Att : \textsc{a}, Lab : n\rangle$
$\quad\longrightarrow$
$\langle O : Ob \mid Pr : (t := n; \textsc{s}), Lvar : \textsc{l}, Att : \textsc{a}, Lab : Next(n)\rangle$
$invoc(eval(x, (\textsc{a}; \textsc{l})), m, (n\ O\ eval(In, (\textsc{a}; \textsc{l}))))$

where x is an object expression, and *eval* is a function which evaluates an expression (list) in the context of a state. In the case of an asynchronous call without an explicit label t, the assignment to t is omitted.

Similarly, a local asynchronous static call $t!m@C'(In)$ gives rise to the invocation message

$invoc(O, m@C', (n\ O\ eval(In, (\textsc{a}; \textsc{l}))))$

and the virtual local call $t!m(In)$ is handled as $t!m@C(In)$ where C is the class of O. A synchronous call, remote or local, virtual or static, is handled by means of an asynchronous call and a reply command:

$\langle O : Ob \mid Pr : p(In; Out); \textsc{s}, Lab : n\rangle$
$\quad = \langle O : Ob \mid Pr : !p(In); n?(Out); \textsc{s}, Lab : n\rangle$

which results in an *invoc* message as defined above.

Transport rules take charge of the message, which eventually arrives at the callee's external message queue. After method execution, a completion message is emitted and eventually arrives at the caller's external message queue.

The caller may wait for a completion in a reply command (including synchronous calls) or in a guard. The reply command blocks until the appropriate reply message has arrived in the external message queue.

$\langle O : Ob \mid Pr : (t?(\textsc{v}); \textsc{s}), Lvar : \textsc{l}\rangle \langle O : Qu \mid Ev : Q\ comp(n, O, Out)\rangle$
$\quad\longrightarrow$
$\langle O : Ob \mid Pr : (\textsc{v} := Out; \textsc{s}), Lvar : \textsc{l}\rangle\ \langle O : Qu \mid Ev : Q\rangle$
if $n = eval(t, \textsc{l})$

A reply guard $t?$ evaluates to true when the *comp* message has arrived, otherwise the active process is put on the internal process queue (see below).

4) Virtual and static binding of method calls: When the invocation of a method m is found in the external message queue of an object O of class C, a message $bind(O, m, In, C)$ is generated where *In* is the actual in-parameter list. Virtual calls are handled by the following equation:

$\langle O : Ob \mid Cl : C\rangle\ \langle O : Qu \mid Ev : Q\ invoc(O, m, In)\rangle$
$\quad = \langle O : Ob \mid Cl : C\rangle\ \langle O : Qu \mid Ev : Q\rangle\ bind(O, m, In, C)$

Static method calls are generated without inspecting the *actual* class of the callee, thus surpassing local definitions:

$invoc(O, m@C', In) = bind(O, m, In, C')$.

If m is defined locally in C, a process with the method code and local state is returned in a *bound* message. Otherwise, the *bind* message is retransmitted to the superclasses of C in a left-first, depth-first order.

$bind(O, m, In, nil) = bound(O, none)$
$bind(O, m, In, (C\ S'))\langle C : Cl | Inh : S, Mtds : \textsc{m}\rangle$
$\quad = \textbf{if}\ (m\ \textbf{in}\ \textsc{m})\ \textbf{then}\ bound(O, get(m, \textsc{m}, In))\ \textbf{else}$
$\qquad bind(O, m, In, (S\ S'))\ \textbf{fi}\ \langle C : Cl \mid Inh : S, Mtds : \textsc{m}\rangle$

The auxiliary function *get* fetches method m in the method multiset M of the class, and returns a process with the method's code and local state. Values of the actual in-parameters *In*, the caller O', and the label value n are stored as local (read-only) variables.

The process resulting from binding a synchronous call is loaded as active code, defined by the following equation:

$bound(O, \langle \textsc{s}', ((label \mapsto n)\ (caller \mapsto O)\ \textsc{l}')\rangle)$
$\langle O : Ob \mid Pr : (n?(\textsc{v})); \textsc{s}, PrQ : W, Lvar : \textsc{l}\rangle =$
$\quad \langle O : Ob \mid Pr : \textsc{s}'; cont(n), PrQ : ((n?(\textsc{v})); \textsc{s}, \textsc{l})\ W, Lvar : \textsc{l}'\rangle$

The additional command $cont(n)$ ensures that only the process which made the call may continue after method completion, thereby causing a LIFO discipline on *PrQ* for local synchronous calls. For an asynchronous call the resulting process R is loaded into the internal process queue, defined by the equation

$$bound(O, R)\ \langle O : Ob \mid PrQ : W\rangle = \langle O : Ob \mid PrQ : R\ W\rangle$$

Here, the last equation only applies if the first equation did not give a match. Note that the use of equations enables the binding mechanism to execute in zero rewrite steps!

5) Guarded Commands: There are three types of guards representing potential processor release points: boolean expressions, wait guards, and return guards. Only evaluation rules for active process return guards are presented here.

Return guards allow process suspension when waiting for method completions, so the object may attend to other tasks while waiting. A return guard evaluates to true if the external message queue contains the completion of the method call, and execution of the process continues. If the message is not in the queue, the active process is suspended. The object can then compute other enabled processes while it waits for the completion of the method call.

$\langle O : Ob \mid Pr : (await\ g?; \textsc{s}), Lvar : \textsc{l}\rangle\ \langle O : Qu \mid Ev : Q\rangle$
$\quad\longrightarrow$
if $inqueue(eval(g, \textsc{l}), Q)$ **then** $\langle O : Ob \mid Pr : \textsc{s}, Lvar : \textsc{l}\rangle$ **else**
$\langle O : Ob \mid Pr : \varepsilon, PrQ : (W\ \langle (await\ g?; \textsc{s}), \textsc{l}\rangle), Lvar : \varepsilon\rangle$ **fi**
$\langle O : Qu \mid Ev : Q\rangle$

where the function *inqueue* checks whether the completion with the given label value is in the message queue Q.

When no process is active, the return guard of the suspended process may be retested against the external message queue. If the completion message is present, the process is reactivated.

$\langle O : Ob \mid Pr : \varepsilon, PrQ : \langle await\ g?; \textsc{s}, \textsc{l}'\rangle\ W, Lvar : \textsc{l}\rangle$
$\langle O : Qu \mid Ev : Q\rangle$
$\quad\longrightarrow$
$\langle O : Ob \mid Pr : \textsc{s}, PrQ : W, Lvar : \textsc{l}'\rangle\ \langle O : Qu \mid Ev : Q\rangle$
if $inqueue(eval(g, \textsc{l}), Q)$

Otherwise, another suspended process from the process queue *PrQ* may be loaded into *Pr*. Remark that any occurrence of a *wait* in a guard causing process suspension is removed.

6) Object Creation: A new object with a unique identifier and an associated event queue are created, after which a synchronous call is made to *run* (if present in the class). New object identifiers are created by concatenating tokens n from the unbounded set *Tok* to the class name. The identifier is returned to the object which initiated the object creation.

$$\begin{array}{l}\langle O : Ob \mid Pr : v := \textit{new}\ C(\textit{In}); \text{S}, \textit{Lvar} : \text{L}, \textit{Att} : \text{A}\rangle\\ \langle C : Cl \mid \textit{Att} : \text{A}', \textit{Tok} : n\rangle\\ \quad\longrightarrow\\ \langle O : Ob \mid Pr : v := \textit{newid}; \text{S}, \textit{Lvar} : \text{L}, \textit{Att} : \text{A}\rangle\\ \langle \textit{newid} : Ob \mid Cl : C, Pr : \textit{run}, PrQ : \varepsilon, \textit{Lvar} : \varepsilon, \textit{Att} : \varepsilon, \textit{Lab} : 1\rangle\\ \langle \textit{newid} : Qu \mid Ev : \varepsilon\rangle\ \langle C : Cl \mid \textit{Att} : \text{A}', \textit{Tok} : \textit{Next}(n)\rangle\\ \textit{find}(\textit{newid}, C(\textit{eval}\,(\textit{In}, (\text{A}, \text{L}))), (\textit{this} \mapsto \textit{newid}))\end{array}$$

Here, *newid* denotes the new identifier. Class parameters are stored among object attributes. A *find* message, which takes an object identifier, a class inheritance list, and a substitution as arguments, causes the inheritance tree to be traversed in a left-first depth-first order, in order to dynamically accumulate and initiate all inherited attributes, while passing on appropriate class parameters as stated in the inheritance list. The completed traversal results in a message *found*, with the object identifier and a substitution (i.e. a local state) as arguments.

$$\begin{array}{l}\textit{find}(O, \textit{nil}, \text{A}) = \textit{found}(O, \text{A})\\ \textit{find}(O, ((C(\textit{In}))\ \text{S'}), \text{A})\ \langle C : Cl \mid \textit{Inh} : \text{S}, \textit{Att} : \text{IA}\rangle\\ \quad = \textit{find}(O, (\text{S S'}), (\text{A}\ \ \textit{initeval}(\text{IA}, \textit{In}, \text{A})))\ \langle C : Cl \mid \textit{Inh} : \text{S}, \textit{Att} : \text{IA}\rangle\end{array}$$

We here denote by IA a state where variables are bound to expressions and not only data values. The auxiliary function *initeval* uses a state A to evaluate (sequentially from left to right) attributes initialized by expressions in IA while passing the parameters *In*. The resulting state is consumed by the object requesting *find* by the equation

$$\textit{found}(O, \text{A})\ \langle O : Ob \mid \textit{Att} : \varepsilon\rangle = \langle O : Ob \mid \textit{Att} : \text{A}\rangle$$

Notice again that the use of equations enables a new object to be created and initialized in a single rewriting step.

7) Testing Specifications in the Creol Interpreter: Specifications in RL are executable on the Maude modeling and analysis tool [5]. This makes RL well-suited for experimenting with programming constructs and language prototypes, combined with Maude's rewrite strategies and search and model-checking abilities. Development and testing of language constructs can be done incrementally. The operational semantics described in this paper has been used as a language interpreter to analyze Creol models [30]. The interpreter consists of 700 lines of code, including auxiliary functions and equational specifications, and it has 25 rewrite rules.

Although the proposed operational semantics is highly non-deterministic, Maude rewriting is deterministic in its choice of which rule to apply to a given configuration. For the evaluation of specifications of non-deterministic systems in Maude, as targeted by Creol, this limitation restricts the direct applicability of the tool as every run of the specification will be identical. However, RL is reflective [31], which allows execution strategies for Maude programs to be written in RL. A strategy based on a pseudo-random number generator is proposed in [30]. Using this strategy, it is easy to test a specification in a series of different runs by providing different seeds to the random number generator. By executing the operational semantics, Maude may be used as a model analysis tool. Maude's search and model checking facilities can be employed to look for specific configurations or configurations satisfying given conditions.

VI. Related Work

Many object-oriented languages offer constructs for concurrency. A common approach has been to rely on the tight synchronization of RPC and keep activity (threads) and objects distinct, as done in Hybrid [6] and Java [7], or on the rendezvous concept in concurrent objects languages such as Ada [2] and POOL-T [32]. For distributed systems, with potential delays and even loss of communication, these approaches seem less desirable. Hybrid offers *delegation* to (temporarily) branch an activity thread. Asynchronous method calls can be implemented in e.g. Java by explicitly creating new threads to handle calls [33]. UML offers asynchronous event communication and synchronous method invocation but does not integrate these, resulting in significantly more complex formalizations [34] than ours. To facilitate the programmer's task and reduce the risk of errors, implicit control structures based on asynchronous method calls seem more attractive, allowing a higher level of abstraction in the language.

Languages based on the Actor model [11], [12] take asynchronous messages as the communication primitive for loosely coupled processes. This makes Actor languages conceptually attractive for distributed programming. Representing method calls by asynchronous messages has lead to the notion of future variables found in e.g. ABCL [21], Eiffel// [23], CJava [33], and in the Join-calculus [35] based languages Polyphonic C$^\sharp$ [24] and JoinJava [22]. Our proposed asynchronous method calls resemble future variables, and inner processor release points further extend this approach to asynchrony.

Most languages supporting asynchronous methods either disallow inheritance [21], [22] or impose redefinition of asynchronous methods [23]. In Polyphonic C$^\sharp$ inheritance is expressed as a disjunction of join patterns [35], resulting in nondeterminism rather than overloading, and supplemented by a substitution mechanism for inherited code. CJava [33], restricted to outer guards and single inheritance, allows separate redefinition of synchronization code and bodies in subclasses.

Maude's inherent object concept [4], [5] represents an object's state as a subconfiguration, as we have done in this paper, but in contrast to our approach object behavior is captured directly by rewrite rules. Both Actor-style asynchronous messages and synchronous transitions (rewrite rules which involve more than one object) are allowed, which makes Maude's object model very flexible. However, asynchronous method calls and processor release points as proposed in this paper are hard to represent within this model. Inheritance is by disjoint union of methods, also resulting in nondeterminism.

VII. Conclusion

The idea of the paper is to show how the concepts of asynchronous method calls and multiple inheritance can be integrated in the setting of distributed concurrent objects in a smooth manner, including asynchronous local calls. The approach allows active and passive behavior to be easily combined in concurrent objects. Previous approaches have not combined asynchronous communication with inheritance in a satisfactory manner. This idea is materialized through a small language Creol with an executable operational semantics based on rewriting logic. A peer-to-peer example demonstrates the suitability of the language constructs in a distributed setting. The major parts of the operational semantics concerning virtual and static binding, synchronous and asynchronous method calls, and object creation, are presented, ignoring aspects of type analysis and semantic requirement specification. In particular, we have given an operational semantics of inheritance and virtual binding based on dynamic and distributed traversal of the available classes, rather than statically given inheritance trees. Our approach may therefore be combined with dynamic constructs for changing the class inheritance structure, such as adding a class C and enriching an existing class with C as a new superclass, which could be useful in open reconfigurable systems.

We have demonstrated through examples how multiple inheritance combined with synchronous merge and synchronous local calls may reduce the inheritance anomaly, allowing mix-in classes and an aspect oriented programming style. It remains to investigate how our approach could be integrated in standard languages and technologies, and to what extent more elaborate programming environments would interfere with the clean computational model proposed in this paper.

REFERENCES

[1] International Telecommunication Union, "Open Distributed Processing - Reference Model parts 1–4," ISO/IEC, Geneva, Tech. Rep., July 1995.

[2] G. R. Andrews, *Concurrent Programming: Principles and Practice*. Reading, Mass.: Addison-Wesley, 1991.

[3] E. B. Johnsen and O. Owe, "An asynchronous communication model for distributed concurrent objects," in *Proc. 2nd IEEE Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*. IEEE Computer Society Press, Sept. 2004, pp. 188–197.

[4] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical Computer Science*, vol. 96, pp. 73–155, 1992.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, "Maude: Specification and programming in rewriting logic," *Theoretical Computer Science*, vol. 285, pp. 187–243, Aug. 2002.

[6] O. Nierstrasz, "A tour of Hybrid – A language for programming with active objects," in *Advances in Object-Oriented Software Engineering*, D. Mandrioli and B. Meyer, Eds. Prentice Hall, 1992, pp. 167–182.

[7] J. Gosling, B. Joy, G. L. Steele, and G. Bracha, *The Java language specification*, 2nd ed., Reading, Mass.: Addison-Wesley, 2000.

[8] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing, "An event-based structural operational semantics of multi-threaded Java," in *Formal Syntax and Semantics of Java*, ser. LNCS, J. Alves-Foss, Ed. Springer, 1999, vol. 1523, pp. 157–200.

[9] E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen, "Verification for Java's reentrant multithreading concept," in *Intl. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'02)*, ser. LNCS, vol. 2303. Springer, Apr. 2002, pp. 5–20.

[10] P. Brinch Hansen, "Java's insecure parallelism," *ACM SIGPLAN Notices*, vol. 34, no. 4, pp. 38–45, Apr. 1999.

[11] G. A. Agha, "Abstracting interaction patterns: A programming paradigm for open distributed systems," in *Proc. 1st IFIP Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'96)*, E. Najm and J.-B. Stefani, Eds. Chapman & Hall, 1996, pp. 135–153.

[12] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, no. 1, pp. 1–72, Jan. 1997.

[13] N. Carriero and D. Gelernter, "Linda in context," *Communications of the ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[14] P. America, "Designing an object-oriented programming language with behavioural subtyping," in *Foundations of Object-Oriented Languages*, J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds. Springer, 1991, pp. 60–90.

[15] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811–1841, Nov. 1994.

[16] N. Soundarajan and S. Fridella, "Inheritance: From code reuse to reasoning reuse," in *Proc. Fifth Intl. Conf. on Software Reuse (ICSR5)*, P. Devanbu and J. Poulin, Eds. IEEE Computer Society Press, 1998, pp. 206–215.

[17] E. B. Johnsen and O. Owe, "A compositional formalism for object viewpoints," in *Proc. 5th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*, B. Jacobs and A. Rensink, Eds. Kluwer Academic Publishers, Mar. 2002, pp. 45–60.

[18] ——, "Object-oriented specification and open distributed systems," in *From Object-Orientation to Formal Methods: Dedicated to the Memory of Ole-Johan Dahl*, ser. LNCS, O. Owe, S. Krogdahl, and T. Lyche, Eds. Springer, 2004, vol. 2635, pp. 137–164.

[19] S. Matsuoka and A. Yonezawa, "Analysis of inheritance anomaly in object-oriented concurrent programming languages," in *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner, and A. Yonezawa, Eds. The MIT Press, 1993, pp. 107–150.

[20] G. Milicia and V. Sassone, "The inheritance anomaly: ten years after," in *Proceedings of the 2004 ACM Symposium on Applied Computing*. ACM Press, 2004, pp. 1267–1274.

[21] A. Yonezawa, *ABCL: An Object-Oriented Concurrent System*, ser. Series in Computer Systems. The MIT Press, 1990.

[22] G. S. Itzstein and M. Jasiunas, "On implementing high level concurrency in Java," in *Proc. 8th Asia-Pacific Computer Systems Architecture Conf. ACSAC 2003*, ser. LNCS, A. Omondi and S. Sedukhin, Eds., vol. 2823. Springer, 2003, pp. 151–165.

[23] D. Caromel and Y. Roudier, "Reactive programming in Eiffel$/\!/$," in *Proceedings of the Conference on Object-Based Parallel and Distributed Computation*, ser. LNCS, J.-P. Briot, J. M. Geib, and A. Yonezawa, Eds. Springer, 1996, vol. 1107, pp. 125–147.

[24] N. Benton, L. Cardelli, and C. Fournet, "Modern concurrency abstractions for C$\sharp$," in *Proc. 16th European Conf. on Object-Oriented Programming (ECOOP 2002)*, ser. LNCS, B. Magnusson, Ed., vol. 2374. Springer, June 2002, pp. 415–440.

[25] J. Dovland, E. B. Johnsen, and O. Owe, "Reasoning about asynchronous method calls and inheritance," submitted for publication, 2004.

[26] E. Najm and J.-B. Stefani, "A formal semantics for the ODP computational model," *Computer Networks and ISDN Systems*, vol. 27, pp. 1305–1329, 1995.

[27] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, "Introducing OBJ," in *Applications of Algebraic Specification using OBJ*, J. Goguen, Ed. Cambridge, 1993.

[28] C. de Oliveira Braga, "Rewriting logic as a semantic framework for modular structural operational semantics," Ph.D. dissertation, Pontifícia Universidade Católica do Rio de Janeiro, 2001.

[29] O. Owe and I. Ryl, "Reasoning control in presence of dynamic classes," in *12th Nordic Workshop on Programming Theory*, Bergen, 2000, online proceedings http://www.ii.uib.no/$\sim$nwpt00.

[30] E. B. Johnsen, O. Owe, and E. W. Axelsen, "A run-time environment for concurrent objects with asynchronous methods calls," in *Proc. 5th Intl. Workshop on Rewriting Logic and its Applications (WRLA'04)*, ser. Electronic Notes in Theoretical Computer Science. Elsevier, Mar. 2004, to appear.

[31] M. Clavel, *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.

[32] P. America, "POOL-T: A parallel object-oriented language," in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds. Cambridge, Mass.: The MIT Press, 1987, pp. 199–220.

[33] G. Cugola and C. Ghezzi, "CJava: Introducing concurrent objects in Java," in *4th Intl. Conf. on Object Oriented Information Systems*, M. E. Orlowska and R. Zicari, Eds. Springer, 1997, pp. 504–514.

[34] W. Damm, B. Josko, A. Pnueli, and A. Votintseva, "Understanding UML: A formal semantics of concurrency and communication in Real-Time UML," in *First Intl. Symposium on Formal Methods for Components and Objects (FMCO 2002), Revised Lectures*, ser. LNCS, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 2852. Springer, 2003.

[35] C. Fournet, C. Laneve, L. Maranget, and D. Rémy, "Inheritance in the join calculus," in *20th Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2000)*, ser. LNCS, S. Kapoor and S. Prasad, Eds., vol. 1974. Springer, Dec. 2000, pp. 397–408.